Software Evaluation

Anne Canteaut, Miguel Angel Fernández, Luc Maranget, Sophie Perin, Mario Ricchiuto, Manuel Serrano, Emmanuel Thomé

Commission d'Evaluation

Software Evaluation

Prepared by the working group "Software Evaluation"
of the following members of the Evaluation Committee:
Anne Canteaut, Miguel Fernandez, Luc Maranget, Sophie Perin,
Mario Ricchiuto, Manuel Serrano (coordinator), Emmanuel Thomé

Approved by the Inria Evaluation Committee on January 14, 2021

January 2021

This document is intended for the members of Inria juries in order to assist them in their evaluation of software developed by the researchers. It results from the work carried out by the Software Evaluation working group from March to December 2020.

1 Foreword

Research candidates for the various Inria recruitment and promotion competitions are requested to qualify any software developments they may have produced, using the terminology set out in the document entitled "Criteria for self-assessment of Software V2", dated June 2011. Experience gained through taking part in these juries has taught us that although the document provides valuable assistance to the applicants, it is often not used appropriately by the juries. The aim of this present document is to rectify this situation. Intended for assessors (whether they are members of a jury or serving on a panel of experts), it should serve as a complement to the document given to applicants and teams. In order to have two complementary, coherent documents, it was necessary to make several changes to the self-assessment document. The new version, which replaces the previous one, is entitled "Criteria for Software Self-Assessment V3".

The first point that we feel should be stressed is that the self-assessment document does not, in itself, constitute an evaluation of the software - all the more so given that it is written by the applicants themselves. On the other hand, applicants should be aware that this document gives them the opportunity to highlight certain aspects of their software and to specify by which criteria they wish to be evaluated. It is then up to the jury to proceed with this evaluation.

The second point is that software evaluation, like any other scientific or research activity, cannot be carried out according to purely syntactic and algorithmic criteria. An applicant's reviewers are therefore asked to give a personal opinion that will, inevitatably, contain a degree of subjectivity, as is indeed the case for reviewing articles. This opinion should be based on sound arguments, and not simply a repetition of a self-assessment produced by the applicant. The evaluation should therefore be based on what the software actually *is* and so the first aspect to examine is the source code, possibly with the help of peer reviews (which

are becoming increasingly used in many domains), or, if the software cannot be publicly distributed, through detailed user or sponsor notices, or results

obtained from the software. The evaluation should also cover the software's diffusion, its documentation, stability, the investment in terms of the time needed for its development, and elements relating to its use. As for the many other aspects developed in a file for promotion or application, the reviewers should assess whether software development is, or is not, an important aspect of the file.

The purpose of this document is to provide guidance on how to carry out the evaluation. It is done in a relatively informal way by listing questions or points that may be raised by the jury during its evaluation. The central element that the evaluation should focus on, independently of the type of development, is the importance of software development in the applicant's file. For some applicants, development is purely anecdotal and does not require any special attention by the jury. For other applicants, however, software development and its diffusion are central to the research activity, and the jury should be particularly attentive to the quality and the role played by development in the applicant's research.

2 Assessing the applicant's role

Whatever the type of software, the first objective will be to establish the role of the applicant. This can be done by focusing on two aspects: the nature of the applicant's contributions, and the duration of the applicant's involvement. Here we make a distinction between:

- anecdotal participation (simply creating several examples, making some corrections, preparing tutorials or parts of the documentation);
- core participation in the development (writing essential components) ;
- participation in the diffusion (writing installation scripts, deployment scripts, etc.).
- ...

Finally, if the software has been created by several authors, the assessors will be asked to justify the nature of the contribution using factual arguments.

In this document, we recommend evaluating software development only for candidates that are directly involved in the development, *i.e.*, in the writing of the code. It is worth mentioning here that we consider that the term "architect" – so often used by researchers – most frequently refers to a supervisory role, either for students or an engineer. This is an important function, but it is one that should be mentioned in the appropriate section devoted to supervision and not in the development section. Similarly, transfer activities, such as implementing or facilitating a consortium linked to software should not be considered as a development activity, but as a transfer activity.

Just as someone would only be considered to be the author of a publication if his name appears clearly at the head of the text, a researcher can only be considered to be the author of a software application if there are tangible traces of his personal involvement in its development (his name presents in the source or in the history of version management systems).

The role of software architect, in its more industrial sense[1], lies halfway between supervision and development, and is therefore somewhat difficult to define with any degree of precision. In general terms, the software architect must organize the development of a software package in such a way as to respect the specification that satisfies user requirements. The development is generally spread over an extended period and the components that constitute it are many and frequently developed by different teams. Some software coming from academic research does indeed fall into this category. The aim of the evaluation will therefore be to establish whether the complexity of the software developed can justify a researcher's claims to be the architect and, if so, to assess the applicant's investment in this activity.

Finally, another essential aspect of the evaluation will be to consider the relevance of the development in an applicant's file. Here are some examples of questions that we might expect to find answered in the evaluation reports. *Was the development conducted in a manner coherent with the applicant's academic research? Are the objectives of the development clear and relevant? Does the development justify subsequent development or, on the contrary, has the development been oversized with regard to the scientific objectives that were sought? Has the development achieved its goals?*

3 Characterizing the type of software

Software applications form a hetereogeneous population. They can be large-scale or small-scale. Some are extremely complex and take years to complete, while others can be more or less jotted down in an afternoon to satisfy a particular need at a particular time. There are some whose realization is particularly difficult and requires a very detailed development, and others whose difficulty lies in seeking previously obtained scientific developments. Some are developed as tools to obtain scientific results, whereas others result from research to prove, for instance, the existence of an algorithmic solution to a given problem. The second task of the evaluation is therefore to *characterize* the software. To do so, we propose to classify them in four major categories. Of course, these categories may overlap and there might be software that belongs to more than one category. There may also be software whose status changes over time.

The characterization of the software will have to be compared with the contributions of the candidate because some software benefits from contributions of a very different nature. This will probably be the case for complex software that has been developed by large teams. In such cases, it is conceivable that some of the contributors focused on the *scientific* development, whereas others focused more on the aspects of *practical implementation*. The individual contributions will therefore have to assessed for what they are and the place they occupy in the person's background and application.

[1] According to Wikipedia (`https://fr.wikipedia.org/wiki/Architecte_logiciel`), and translated here from the French,: a software architect is "*a computer expert who is responsible for creating and maintaining the software architecture model. This is distinct from the computer architect who works on hardware. A software architect is a certified professional computer scientist or a software engineer who, in several countries, is a member of a professional body. In small projects, the role of the software architect can be held by the analyst, the project manager or the developer in charge of the project. Large companies may have a Chief Software Architect who is responsible for the application of architectural standards to all projects, the management of the project, and the reuse of the company's software components. In large-scale projects, it is possible to find a software architect and several software sub-architects, each responsible for the separate parts of the development. The functional analyst, the software architect and the software engineer have distinct responsibilities: the first pilots the use cases, the second the architecture and the third the development.*"

3.1 Software as a Vector for Knowledge

Here we consider software applications that have a dual purpose. On the one hand, they perform a specific task corresponding to a clear scientific objective and which gives the software its value. On the other hand, they communicate knowledge as a scientific object: either because examining their construction is instructive, or because they allow other researchers to build variants that will allow other avenues of research to be explored. The first aspect is important because it justifies the interest of the software's existence. The second aspect is crucial as it gives the software its status as a scientific object.

Consequently, for software to be included in this category, **its source code must necessarily be publicly available** because it is there that, for the most part, the disseminated knowledge is to be found.

For software to be considered as belonging to this "Vector for Knowledge" category, it must satisfy one or more of the criteria given in the following (*non-exhaustive*) list:

- the software is based on the results of research that has been made available through traditional means such as conference papers, scientific journals, books, etc.;
- the development of this software feeds a research activity traditionally described in scientific papers and gives rise to further software developments;
- the development required new scientific advances to be made;
- the software was mainly developed by one or more researchers (possibly, of course, with the help of software engineers);
- it was developed with the aim of being disseminated, and therefore benefits from a website that presents it, with the appropriate documentation, and describes how to install it;
- the source code is publicly available (gitlab, github, or another). The development history, provided by version control systems, is of great importance to understand the dynamic of the software evolution.

3.1.1 Evaluation criteria

Here are some criteria to consider when assessing and comparing software in this category.

- Scientific complexity of the problem addressed.
- Complexity of the development (size of the code, hardware specificities, complex porting on esoteric machines, intrinsic computing difficulty, size of the development team.
- Links between the software and scientific publications (how software development and academic results feed into each other, similarly, though perhaps to a lesser degree, for PhD theses, assessing software in conferences that carry out the evaluation of artifacts, etc.)
- Quality of the development (use of good practices, state-of-the-art development methods, the quality of the code, etc.).

- Quality of the distribution (website, documentation, ease of installation, packaging, etc.).
- Impact (the existence of a community of users attested by external contributions, citations in other studies, inclusion in other open systems, etc.).
- ...

3.1.2 Examples

- Coq, `https://coq.inria.fr/`: formal proof management system. The development of Coq required many advances in the field of programming languages. Its evolution centers around the concept of mathematical proof, and its scientific impact is apparent in both computer science and mathematics.
- FreeFem, `https://freefem.org/`: an open-source software to solve partial differential equations using the finite element method. It has its own script language, inspired by C++, to facilitate the writing of the variational formulation of boundary problems with partial differential equations. The platform was developed to support learning and basic research through prototyping.
- SimGrid, `https://simgrid.org/`: a framework for developing simulators of distributed applications. It has allowed numerous experimental results over a long period of time and has supported the research in hundreds of scientific publications.
- FEniCS, `https://fenicsproject.org`: an open-source platform for solving partial differential equations using the finite element method. It has a Python interface to facilitate writing the variational formulation of PDEs and their resolution using parallelism via the interface with external libraries (e.g. PETSc).
- GNU MPFR, `https://www.mpfr.org/`: an open-source library for multiple-precision floating-point computations with correct rounding. GNU MPFR had a major impact on the developments of the IEEE-754 standard as well as on many software projects that either used it as a building block or extended it, inspired by the structure of the algorithms used in GNU MPFR (libraries offering extensions to C or to polynomials).
- Although the five examples above can be described as large-scale software projects, this category may also include smaller projects such as qDSA (`https://joostrenes.nl/software.html`) or relic-toolkit (`https://github.com/relic-toolkit`). In these cases, the interest does not lie so much in what the software achieves, as in the way it does so: the source code of the software, which meets the small-scale objectives, and the execution time and performance are essential objects of study.
- Unison, `https://www.cis.upenn.edu/~bcpierce/unison`: a tool for file synchronization. It is an interesting example because it began as a tool for research in advanced distributed computing. It later became used as utility software. Today, Unison is no longer used for research in distributed programming, but remains widely used for synchronization between machines.

3.2 Software as a Vehicle for Research

This section deals with software that is closely linked to research but which does not, in itself, represent a scientific result. The main purpose of "Software Infrastructure" is often to validate scientific work.

Such software can be placed in one of two categories. The first is for software that is used to validate work or to make it reproducible. This usually concerns small programs that are written to validate an experiment or an algorithm described in a scientific paper. Sometimes, the care taken to develop the software is minimal since its life-span and purpose are limited. Other times, the care taken is greater as the purpose may go beyond simply reproducing an experiment. This is the case, for instance, when its use can be generalized and the software can be used to carry out a whole series of experiments.

The second category is made up of software platforms, *i.e.*, the sets of software building blocks (routines, data structure, input/output interfaces, etc.), that are used to bring together the developments of several researchers, or even of one or more research teams. These platforms provide a framework that allows new researchers, for example PhD students, to contribute to projects in a way that exceeds what they could achieve alone.

3.2.1 Evaluation criteria

Here are some criteria to consider when assessing and comparing software in this category.

- If the only purpose of the software is to validate an experiment, then the evaluation of the software itself needs not go into great depth: it should focus more on the link between the academic result obtained and the possible difficulties of the development. In such a case, the validation and reproducibility of the results are important. The software should be available and documented at least to the degree that such reproducibility is possible

- If the software is intended to be used by other researchers, then the evaluation should be more precise. It should provide some indication of its impact (number of users, how it has made collaboration possible) and its ease of use (appropriate documentation, ease of installation, etc.).

- For the case of software platforms, the difficulty of the development should be assessed as well as the size of the problem addressed. The evaluation should provide an indication of the scale and duration of the development, and it should establish whether the platform makes it possible to address a particular problem with very specialized solutions or whether it can be used to address a broad family of problems.

- Other evaluation criteria may, of course, be relevant. For example, some software applications have strong and complex dependencies with other software tools, which makes their development more difficult, and this should be taken into account in the evaluation.

- The pedagogical aspect, and particularly the integration of PhD work in the software development should also be considered.

3.3 Transfer Software

This mainly concerns software developed for industrial partners. The code is not usually publicly available and the development generally involves an appropriate operating licence.

3.3.1 Evaluation criteria

- The use of the software must be confirmed by one or more industrial companies (the software is mentioned on a website, a statement or recommendation by industrial users, a description of which new technological solutions are provided, significant licences paid for by the companies involved, etc.).
- Complexity of the problem addressed.
- Importance of the scientific results that are transferred through the software.
- Complexity of the development (details concerning the size of the code, the implementation, the implementation language, duration of the development, etc.).
- Use of good practices (appropriate and state-of-the-art development methods), existence of a website, documentation and ease of installation (at least documented).
- It is also necessary to specify the relationship between the version actually used by the industrial partners and the software developments carried out by the academic partners.
- ...

3.4 Utility Software

Researchers are sometimes led to develop software that is not related to their own work or that of their colleagues, nor to any transfer activity. Sometimes they contribute to the development of free utility software, or the development of educational packages, or else they simply aim to satisfy a need in their professional environment. Although the development of such software can be time-consuming and complex, its lack of a direct link with any research activity makes it less important in the evaluation of applicants than the other categories of software development.

3.4.1 Examples

Here are some examples of software belonging to this category, produced by members of our institute.

- Flyspell.el, `https://www.gnu.org/software/emacs/`: a minor mode on-the-fly spell-checker in Emacs.
- Gnubiff, `http://gnubiff.sourceforge.net/`: a mail notification program.
- Hevea, `http://hevea.inria.fr/`: a LaTeX to HTML translator.

3.4.2 Evaluation criteria

- Size of the user community.
- Duration of the development and of the software use.
- Quality of Service provided.
- ...

4 Self-Assessment of Software, revised version

This appendix contains a revised version of the document entitled "INRIA Evaluation Committee Proposal of Criteria for Software Self-Assessment".

The EC considers software a major research outcome of the Institute. However, experience has shown that software descriptions in the team descriptions and the candidate application files are very hard to use, because team members and applicants are much less used to self-assessment of software than to self-assessment of theoretical contributions and publications.

The goal of this document is to provide Inria teams and candidates with a self-assessment mechanism for their software developments, to be used by the Inria Evaluation Committee (EC) for evaluation seminars and internal or external recruitments/promotions. It is **a request for evaluation** that the juries and evaluation panels will use to select the appropriate criteria for evaluating software applications developed by teams and candidates. What the EC wants is a fair self-assessment of the real state, evolution, and impact of the software, which in principle would prove exact in a subsequent in-depth evaluation. Examples of some Inria Software Artifacts.

- OCaml: `Family=research; Audience=community; evolution=lts; Duration>=20; contribution=devel,softcont; Url=https://caml.inria.fr/ocaml/`
- Flyspell.el: `Family=utility; Audience=universe; evolution=basic; Duration>=20; contribution=leader; Url=https://www.emacswiki.org/emacs/FlySpell`

The software self-assessment should be followed (Section *Free Description*) by a brief description of the software you have contributed to and the nature of your contributions.

4.1 Software Family (Family)

The software families are described in the EC document "Software Artifact Evaluation", referred to as SAE.

1. `research`: Software as a Vector for Knowledge (see SAE, Section 3.1).
2. `vehicle`: Software as a Vehicle for Research (see SAE, Section 3.2).
3. `transfer`: Transfer software, (see SAE, Section 3.3).
4. `utility`: Utility, (see SAE, Section 3.4).

4.2 Audience (Audience)

1. `personal`: personal or internal team prototype (to experiment an idea);
2. `team`: to be used by people in the team or close to the team (including contractual partners);
3. `partners`: to be used by people inside and outside the team but without a clear and strong dissemination and support action plan;
4. `community`: large audience software, usable by people inside and outside the field with a clear and strong dissemination, validation, and support action plan;
5. `universe`: wide-audience software (aims to be usable by a wide public, to become the reference software in its area, etc.).

4.3 Evolution and Maintenance

- `nofuture`: no real future plans;
- `basic`: basic maintenance to keep the software alive;
- `lts`: long term support.

4.4 Duration of the Development (Duration)

Indicate here the number of years of your contribution to the software development.

4.5 Contribution Characterization

Characterize your contributions to the development. More than one contribution are possible, in particular if the nature of your contribution has evolved over the years.

- `leader`: you have been at the initiative of the development and you have been involved in the coding, debugging, and maintenance.
- `instigator`: you have been at the initiative of the development but you have not been involved in the coding.
- `devel`: you have been deeply involved in the coding.
- `softcont`: you have contributed to the documentation, maintenance, installation scripts, ...

Supervising activities must be reported in dedicated section (Form 1) in the application form.

4.6 Web Page (Url)

Indicate the software artifact URL.

Free Description

In this free section, present in **at most 10 lines**, all the pieces of information you will find useful to let the examiner better understand the software application you have developed and the nature of your contribution. This section can contain facts such as the programming language(s) used, the code size, the complexity and the nature of your contribution, etc. It can also describe the relationships between this software and your other research activities.